21.5.1 Defining New Overloadings for Operators

It is possible to define new overloadings for existing operators or even to hide the existing definiton of one of the standard operators and replace it with a new definition. Suppose in a 3-D graphicsprogram we define:

```
TYPE Vector is ARRAY(1 .. 3) OF Float;
A, B, C, D : Vector;
```

so that objects of type `Vector` can represent the coordinates of points in 3-dimensional space. We cannot use the standard arithmetic operators `"+"`, `"-"` and so on with objects of type `Vector` directly:

```
C : = A + B;      -- !!?? compilation error
```

However we can define new overloadings to make such steps legal in Ada:

```
FUNCTION "+"(Left, Right : Vector) RETURN Vector IS
   Sum : Vector;
BEGIN
   FOR I IN Vector'Range LOOP
      Sum(I) := Left(I) + Right(I);
   END LOOP;
   RETURN Sum;
END "+";
```

Now we can quite legally use the arithmetic operator `"+"` to combine values of type `Vector`:

```
C : = A + B;      -- vector sum
```

Notes

a) The operator name is enclosed in double quotes in its definition, but not when it is used.

b) It is possible to call operators using the standard prefix notation for function calls, for example:

```
C : = "+"(A, B);
```

However such usage would be stylistically rather bizarre!

d) We can only define new overloadings of existing operators, namely:

```
+      -      *      /      **
ABS    &      REM    MOD  (modulus similar to REM)
>      <      >=     <=     =      /=
NOT    AND    OR     XOR  (exclusive or)
```

`ABS` and `NOT` are unary prefix operators whilst all the rest are binary infix operators although `"+"` and `"-"` may also be used as unary prefix operators. Note that the list does not include `IN`, `NOT IN`, `AND THEN` and `OR ELSE` which are not technically operators.

For example we could define a new overloading of `ABS` to give the length of a vector:

```
FUNCTION "ABS"(Right : Vector) RETURN Float IS
   SumSquares : Float := 0.0;
BEGIN
   FOR I IN Vector'Range LOOP
      SumSquares := SumSquares + Right(I)**2;
   END LOOP;
   RETURN Sqrt(SumSquares);
   -- Uses Sqrt from Ada.Numerics.Elementary_Functions
END "ABS";
```

This could then be used as follows:

```
Put("The length of vector A is "); Put(ABS A);
```

c) The number of arguments and the priority (or precedence) level for new operator overloadings must be the same as those of the standard operators. Thus any overloadings of `"*"` and `"/"` must be binary operators (i.e. have two operands). These overloadings have a higher precedence than `"+"` and `"-"` but lower precedence than `"**"`. Since `"+"`

and "-" can be used as binary infix or as unary prefix operators we can define overloadings of these operators which take either one or two operands; see for example the two overloadings of "-" below defined for the type Vector.

e) New overloadings of the comparison operators ">", "<", "="etc. need not necessarily return a Boolean result, (although overloadings with other return types are likely to cause confusion and so should generally be avoided on stylistic grounds). There is however one important restriction: if an overloading of "=" is defined which does return a Boolean result, then a corresponding overloading of "/=" is automatically created.

f) It is not possible to define default parameter values for operators. The reason for this restriction is that an operator call with missing parameters would look bizarre and would make syntax checking difficult for the compiler.

Normally we would gather together all the overloadings of operators for a new type in a package together with the definition of the type and perhaps some useful constants.

Thus we might have

```
PACKAGE Vectors IS

   TYPE Vector IS ARRAY(1 .. 3) OF Float;

   Zero : CONSTANT Vector := (1 .. 3 => 0.0);  -- zero vector
   -- 3 unit coordinate vectors along x, y and z axes
   I    : CONSTANT Vector := (1  => 1.0, 2|3 => 0.0);
   J    : CONSTANT Vector := (2  => 1.0, 1|3 => 0.0);
   K    : CONSTANT Vector := (3  => 1.0, 1|2 => 0.0);

   FUNCTION "+"(Left, Right : Vector) RETURN Vector;
      -- Vector Addition

   FUNCTION "-"(Left, Right : Vector) RETURN Vector;
      -- Vector Subtraction

   FUNCTION "-"(Right : Vector) RETURN Vector;
      -- Vector Negation (unary minus)

   FUNCTION "*"(Left : Float; Right : Vector) RETURN Vector;
      -- Scalar Multiplication (Vector Rescaling)

   FUNCTION "*"(Left, Right : Vector) RETURN Float;
      -- Scalar or Dot Product

   FUNCTION "ABS"(Right : Vector) RETURN Float;
       -- Length of a Vector
END Vectors;
```

```
WITH Ada.Numerics.Elementary_Functions;
WITH Ada.Numerics.Elementary_Functions;

PACKAGE Vectors IS

FUNCTION "+"(Left, Right : Vector) RETURN Vector IS
   Sum : Vector;
BEGIN
   FOR I IN Vector'Range LOOP
      Sum(I) := Left(I) + Right(I);
   END LOOP;
   RETURN Sum;
END "+";

FUNCTION "-"(Left, Right : Vector) RETURN Vector IS
   Diff : Vector;
BEGIN
   FOR I IN Vector'Range LOOP
```

```
      Diff(I) := Left(I) - Right(I);
   END LOOP;
   RETURN Diff;
END "-";

FUNCTION "-"(Right : Vector) RETURN Vector IS
   Minus : Vector;
BEGIN
   FOR I IN Vector'Range LOOP
      Minus(I) := -Right(I);
   END LOOP;
   RETURN Minus;
END "-";

FUNCTION "*"(Left : Float; Right : Vector) RETURN Vector IS
   Product : Vector;
BEGIN
   FOR I IN Vector'Range LOOP
      Product(I) := Left * Right(I);
   END LOOP;
   RETURN Product;
END "*";

FUNCTION "*"(Left, Right : Vector) RETURN Float IS
   ScalarProduct : Float := 0.0;
BEGIN
   FOR I IN Vector'Range LOOP
      ScalarProduct := ScalarProduct + Left(I) * Right(I);
   END LOOP;
   RETURN ScalarProduct;
END "*";

FUNCTION "ABS"(Right : Vector) RETURN Float IS
   SumSquares : Float := 0.0;
BEGIN
   FOR I IN Vector'Range LOOP
      SumSquares := SumSquares + Right(I)**2;
   END LOOP;
   RETURN Sqrt(SumSquares);
END "ABS";
```

We could then use the package in the standard way:

```
WITH Vectors; USE Vectors;
```

Note that it is usual to include a `USE` clause for a package which exports new operator overloadings, otherwise the operator would need to be called using it fully qualified name thus:

```
C := Vectors."+"(A, B);
```

which is very clumsy and would remove most of the advatages of introducing a new operator overloading. An alternative "half-way" house is to use the `USE TYPE` context clause

```
WITH Vectors; USE TYPE Vectors.Vector;
```

which allows us to use operators in the normal way but requires that any exported procedures, functions, constants etc. still need to be specified in fully qualified form.